Analysis and Mitigation of NoSQL Injections

Aviv Ron, Alexandra Shulman-Peleg, and Anton Puzanov | IBM

NoSQL (not only SQL) data storage systems have become very popular due to their scalability and ease of use. Although NoSQL data stores' new data models and query formats make old attacks, such as SQL injections, irrelevant, they give attackers new opportunities to insert malicious code.

Database security is a critical aspect of information security. Access to enterprise databases grants attackers great control over critical data. For example, SQL injection attacks insert malicious code into the statements the application passes to the database layer. This enables attackers to do almost anything with the data, including accessing unauthorized data and altering, deleting, and inserting data. Although SQL injection exploitation has declined steadily over the years owing to secure frameworks and improved awareness, it remains a high-impact means to exploit system vulnerabilities. For example, Web applications receive four or more Web attack campaigns per month, and SQL injections are the most popular attacks on retailers.[1] Furthermore, SQL injection vulnerabilities affect 32 percent of all Web applications.[2]

NoSQL (not only SQL) is a trending term in modern data stores; it refers to nonrelational databases that rely on different storage mechanisms such as document store, key-value store, and graph. The wide adoption of these databases has been facilitated by the new requirements of modern large-scale applications, such as Facebook, Amazon, and Twitter, which need to distribute data across a huge number of servers. Traditional relational databases don't meet these scalability requirements; they require a single database node to execute all operations of the same transaction.[1]

As a result, a growing number of distributed, NoSQL key-value stores satisfy the scalability requirements of modern large-scale applications. These data stores include NoSQL databases such as MongoDB and Cassandra as well as in-memory stores and caches such as Redis and Memcached. Indeed, the popularity of NoSQL databases has grown consistently over the past several years, and MongoDB is ranked fourth among the 10 most popular databases, as Figure 1 illustrates.

In this article, we provide an analysis of NoSQL threats and techniques as well as their mitigation mechanisms.

NoSQL Vulnerabilities

Like almost every new technology, NoSQL databases lacked security when they first emerged.[3–5] They suffered from a lack of encryption, proper authentication, role management, and fine-grained authorization.[6] Furthermore, they allowed dangerous network exposure and denial-of-service attacks.[3] Today, the situation is better, and popular databases have introduced built-in protection mechanisms.[7]

NoSQL databases use different query languages, which makes traditional SQL injection techniques irrelevant. But does this mean that NoSQL systems are immune to injections? Our study shows that although the security of the query language and drivers has largely improved, there are still techniques for injecting malicious queries. Some works already provide reports of NoSQL injection techniques.[1,3,4] Some initial application-scanning projects have emerged (for example, nosqlproject.com), and the Open Web Application Security Project has published recommendations for testing NoSQL injection code. However, these are only initial results; the problem hasn't been sufficiently studied or received the required attention.

Rank Nov. 2015	Rank Oct. 2015	Rank Nov. 2014	Database management system
1.	1.	1.	Oracle
2.	2.	2.	MySQL
3.	3.	3.	Microsoft SQL Server
4.	4.	↑ 5.	MongoDB +
5.	5.	↓ 4.	PostgreSQL
6.	6.	6.	DB2
7.	7.	7.	Microsoft Access
8.	8.	↑ 9.	Cassandra +
9.	9.	↓ 8.	SQLite
10.	10.	↑ 11.	Redis +

Figure 1. Top 10 most popular databases according to db-engines.com popularity ranking, August 2015. NoSQL (not only SQL) databases among the top 10 are MongoDB, Cassandra, and Redis; all three are growing in popularity.

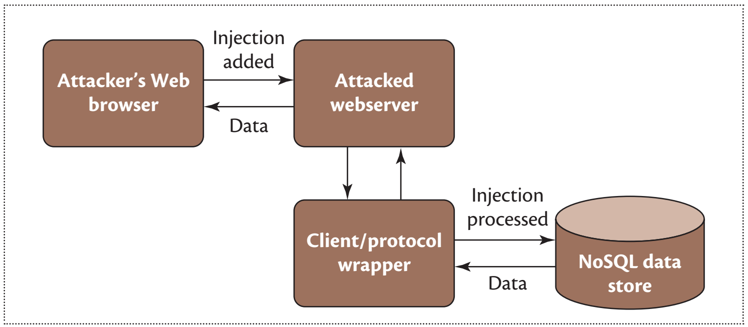


Figure 2. A typical Web application architecture. A NoSQL database is used to store the data accessed via a Web application. Access to the database is performed via a driver—an access protocol wrapper that provides libraries for database clients in multiple programming languages. Although the drivers themselves might not be vulnerable, sometimes they present unsafe APIs that, when used incorrectly by the application developer, could introduce vulnerabilities in the application.

NoSQL Attack Vectors

Web applications and services commonly use NoSQL databases to store customer data. Figure 2 illustrates a typical architecture in which a NoSQL database is used to store the data accessed via a Web application. Access to the database is performed via a *driver*—an access protocol wrapper that provides libraries for database clients in multiple programming languages. Although the drivers themselves might not be vulnerable, sometimes they present unsafe APIs that, when used incorrectly by the application developer, could introduce vulnerabilities in the application that allow arbitrary operations on the database. As Figure 2 shows, attackers can craft a Web access request with an injection that, when processed by the database client/protocol wrapper, will allow the desired illegal database operation.

The main mechanisms of SQL attacks relevant in NoSQL can be divided into five classes.

Tautologies. These attacks allow bypassing authentication or access mechanisms by injecting code in conditional statements, generating expressions that are always true (*tautologies*). For example, in this article, we show how attackers can exploit the syntax of the `$ne` (not equal) operator, which lets them illegally log in to the system without appropriate credentials.

Union queries. Union query is a well-known SQL injection technique in which attackers exploit a vulnerable parameter to change the dataset returned for a given query. The most common uses of union queries are to bypass authentication pages and extract data. In this article, we show an example attack exploiting Boolean `OR` operators by adding expressions that are always true (for instance, an empty query `{}`), which leads to the incorrect evaluation of the entire statement and allows illegal data extraction.

JavaScript injections. This new class of vulnerabilities introduced by NoSQL databases allows execution of JavaScript in the database context. JavaScript enables complicated transactions and queries on the database engine. Passing unsanitized user input to these queries might allow for injection of arbitrary JavaScript code, which could result in illegal data extraction or alteration.

Piggybacked queries. In piggybacked queries, attackers exploit assumptions in the interpretation of escape sequences' special characters (such as termination characters like carriage return and line feed [CRLF]) to insert additional queries to be executed by the database, which could lead to arbitrary code execution by attackers.

Origin violation. HTTP REST APIs are a popular module in NoSQL databases; however, they introduce a new

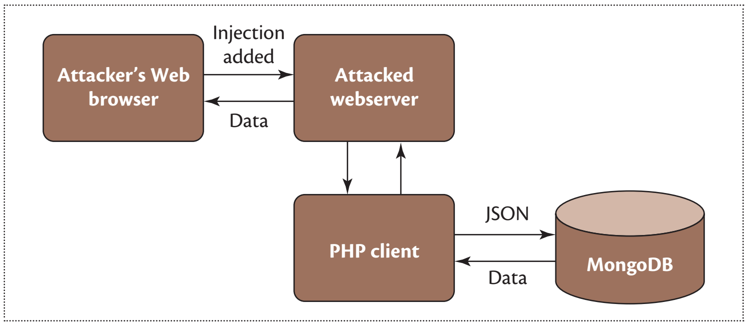


Figure 3. PHP application with MongoDB. A Web application is implemented with a PHP back end, which encodes the requests to the JavaScript Object Notation (JSON) format used to query the data store.

class of vulnerabilities that lets attackers target the database even from another domain. In cross-origin attacks, attackers exploit legitimate users and their Web browsers to perform an unwanted action. In this article, we show such violations in the form of a cross-site request forgery (CSRF) attack in which the trust that a site has in a user's browser is exploited to perform an illegal operation on a NoSQL database. By injecting an HTML form into a vulnerable website or tricking a user into the attacker's own website, an attacker can perform a `post` action on the target database, thus compromising the database.

JavaScript Object Notation Queries and Data Formats

Although relatively safe, the popular JavaScript Object Notation (JSON) representation format allows new types of injection attacks. We illustrate this with an example attack in MongoDB—a document-oriented database that multiple large vendors, including eBay, Foursquare, and LinkedIn, have adopted.

In MongoDB, queries and data are represented in JSON format, which is better than SQL in terms of security because it is more well-defined, is simple to encode and decode, and has good native implementations in every programming language. Breaking the query structure, as has been done in SQL injection, is more difficult with a JSON structured query. A typical insert statement in MongoDB could be the following:

```
db.books.insert({
  title: 'The Hobbit',
  author: 'J.R.R. Tolkien'
  })
```

This inserts a new document into the books collection with a title and author field. A typical query could be

```
db.books.find({ title: 'The Hobbit' })
```

Queries can also include regular expressions and conditions as well as limit which fields are queried.

PHP Tautology Injections

Let's examine the architecture depicted in Figure 3, where a Web application is implemented with a PHP back end, which encodes the requests to the JSON format used to query the data store. Let's use a MongoDB example to show an array injection vulnerability—an attack similar to SQL injection in its technique and results.

PHP encodes arrays to JSON natively. So, for example, the array

```
array('title' => 'The Hobbit',
  'author' => 'J.R.R. Tolkien');
```

would be encoded by PHP to the following JSON:

```
{"title": "The Hobbit", "author":
  "J.R.R. Tolkien"}
```

If a PHP application has a login mechanism in which the username and password are sent from the user's browser via HTTP POST (the vulnerability is applicable to HTTP `get` as well), a typical `post` URL–encoded payload would look like this:

```
username=Tolkien&password=hobbit
```

The back-end PHP code to process it and query MongoDB for the user would look like the following:

```
db->logins->find(array("username"=>$_
  POST["username"],
  "password"=>$_POST["password"]));
```

This makes perfect sense and is intuitively what the developer is likely to do, intending a query of

```
db.logins.find({ username: 'tolkien',
  password: 'hobbit'})
```

However, PHP has a built-in mechanism for associative arrays that lets attackers send the following malicious payload:

```
username[$ne]=1&password[$ne]=1
```

PHP translates this input into:

```
array("username" => array("$[ne] " =>
  1), "password" =>
  array("$ne" => 1));,
```

which is encoded into the MongoDB query

```
db.logins.find({ username: {$ne:1 },
  password {$ne: 1 })
```

Because `$ne` is MongoDB's not equals condition, it queries all entries in the logins collection for which the username is not equal to 1 and the password is not equal to 1. Thus, this query will return all users in the logins collection. In SQL terminology, this is equivalent to:

```
SELECT * FROM logins WHERE username <>
  1 AND password <> 1
```

In this scenario, the vulnerability gives attackers a way to log in to the application without valid credentials. In other variants, the vulnerability might lead to illegal data access or privileged actions performed by an unprivileged user. To mitigate this issue, we need to cast the parameters received from the request to the proper type, in this case, using the string

```
db->logins->find(
  array("username"=>(string)$_
   POST["username"],
  "password"=>(string)$_
   POST["password"]));
```

NoSQL Union Query Injection

SQL injection vulnerabilities are often a result of a query being built from string literals that include user input without proper encoding. The JSON query structure makes attacks more difficult in modern data stores such as MongoDB. Nevertheless, it's still possible.

Let's examine a login form that sends its username and password parameters via an HTTP `post` to the back end, which constructs the query by concatenating strings. For example, the developer would do something like the following:

```
string query = "{ username: '" + post_
  username + "', password:
  '" + post_passport + ' " }"
```

With valid input (tolkien + hobbit), this would build the query:

```
{ username: 'tolkien', password:
  'hobbit' }
```

But with malicious input, this query can be turned to ignore the password and log in to a user account without the password. An example of malicious input is

```
username=tolkien', $or: [ {}, {'a':
  'a&password=' }],

$comment: 'successful MongoDB
  injection'
```

This input would be constructed into the query

```
{ username: 'tolkien', $or: [ {}, {
  'a': 'a', password '' }
], $comment: 'successful MongoDB
  injection' }
```

This query would succeed as long as the username is correct. In SQL terminology, this query is similar to

```
SELECT * FROM logins WHERE username =
  'tolkien' AND (TRUE OR
('a'='a' AND password = ''))
  #successful MongoDB injection
```

The password becomes a redundant part of the query because an empty query `{}` is always true and the end comment doesn't affect the query.

How did this happen? The following constructed query shows user input in bold and the rest in plain text:

```
{ username: 'tolkien', $or: [ {}, {
  'a': 'a', password '' }
], $comment: 'successful MongoDB
  injection' }
```

This attack will succeed in any case in which the username is correct—a valid assumption as harvesting usernames isn't difficult.

NoSQL JavaScript Injection

A common feature of NoSQL databases is the ability to run JavaScript in the database engine to perform complicated queries or transactions such as `MapReduce`. Popular databases that allow this include MongoDB and CouchDB and its descendants, Cloudant and BigCouch. JavaScript execution exposes a dangerous attack surface if unsanitized user input finds its way to the query. For example, consider a complicated transaction that demands JavaScript code and includes unsanitized user input as a parameter in the query. Let's take a model of a store that has a collection of items; each item has a price and an amount. To get the sum or average of these fields, the developer writes a `MapReduce` function that takes the field name that it should act on (`amount` or `price`) as a parameter from the user. In PHP, such code can look like this (where `$param` is user input):

```
$map = “function() {
for (var i = 0; i < this.items.length;
  i++) {
emit(this.name, this.items[i].$param);
   } }”;
$reduce = “function(name, sum) {
  return Array.sum(sum); }”;
$opt = “{ out: ‘totals’ }”;
$db->execute(“db.stores.
  mapReduce($map, $reduce, $opt);”);
```

This code sums the field given by `$param` for each item by name. Then, `$param` is expected to receive either `amount` or `price` for this code to behave as expected. But, because user input isn't being escaped here, a malicious input (that might include arbitrary JavaScript) will execute.

Consider the following input:

```
a);}},function(kv) { return 1; }, {
  out: ‘x’
});db.injection.
  insert({success:1});return
1;db.stores.mapReduce(function() { {
  emit(1,1
```

In the first section, the payload closes the original `MapReduce` function; attackers can then execute any desired JavaScript on the database (in bold). Finally, the last part calls a new `MapReduce` to balance the injected code into the original statement. After combining this user input into the string that gets executed, we get the following (injected user input is in bold):

```
db.stores.mapReduce(function() {
for (var i = 0; i < this.items.length;
  i++) {
emit(this.name, this.items[i].a); }
},function(kv) { return 1; }, { out:
  ‘x’ });
db.injection.insert({success:1});
return 1;db.stores.
  mapReduce(function() { { emit(1,1);
  } },
function(name, sum) { return Array.
  sum(sum); }, { out:
‘totals’ });”
```

This injection looks very similar to classic SQL injections. The defense against such an attack is to disable the use of JavaScript execution in the database configuration. If JavaScript is required, it's best practice not to use any user input in its formation.

Key-Value Data Stores

Key-value data stores such as Memcached, Redis, and Tachyon are in-memory data stores, designed to speed up the performance of applications, cloud infrastructure and platforms, and big data frameworks. These platforms allow for storage and retrieval of data that's accessed repeatedly and frequently (for example, a cache). They're commonly located in front of the data storage, as Figure 4 depicts. Cache platforms often store authentication tokens and container access control lists), which must be revalidated for each subsequent user request.

Although cache APIs are usually very simple because key-value queries are simple, we found a possible injection attack on Memcached, the second most popular key-value store, owing to a vulnerability in the Memcached driver on certain PHP versions. The following conditions must occur to conduct the attack:

- information from the user request (for instance, the HTTP `header`) is used as an attribute (for instance, a value) passed to a cache `set/get` request,
- the received string is passed as is and is not sanitized, and
- cached attributes include sensitive information that will cause the query execution to behave differently than intended.

If these conditions are met, attackers can inject queries or manipulate query logic, such as a piggyback query attack.

Piggybacked Queries

A `set` operation adds a key and its corresponding value to the database using Memcached. When called from the command line interface, the `set` function uses two lines of input. The first is

```
set <KEY> <FLAG> <EXPIRE_TIME>
  <LENGTH>,
```

and the second consists of the data that should be stored.

When the PHP driver's `set` function is called, it receives two parameters and will look like this:

```
$memcached->set(‘key’, ‘value’);
```

Researchers have shown that the driver fails to sanitize the ASCII characters carriage return `\r(0x0D)` and line feed `\n(0x0A)`, resulting in an opportunity for attackers to inject a new line in the key parameter and append another unintended command to the cache.[8]

Consider the following code in which `$param` is user input and is used as the key:

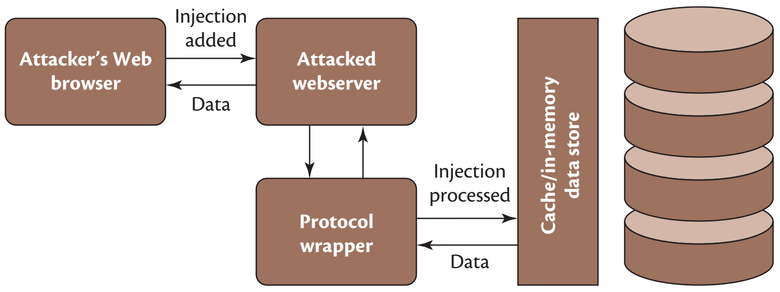


Figure 4. Distributed in-memory data store architecture. The attacked webserver uses a key-value data store for quick data retrieval. Queries to the data store are constructed on the webserver from user-supplied data. If handled wrong, user data can cause an injection of an illegal query that will be processed by the key-value data store and cause a failure in application logic and hence bypass of credentials or unwanted retrieval of data.

```
$memcached=new Memcached();
$memcached
->addServer('localhost',11211);
$memcached->set($param, "some value");
```

Attackers can supply the following input that will result in an injection:

```
"key1 0 3600 4\r\nabcd\r\nset key2 0
3600 4\r\ninject\r\n"
```

In this example, the first key added to the database is `key1`, with the value `some value`. Attackers can add another, unintended key to the database, `key2`, with the value `inject`.

This injection can also take place from the `get` command. Let's examine the example on Memcached's homepage, which starts with these three lines:

```
Function get_foo(foo_id)
foo = memcached_get("foo: " . foo_id)
return foo if defined foo
```

This example shows a typical use of Memcached; it checks whether the input exists in the database before processing it. Assume similar code is used to check authentication tokens received from users to verify whether they're logged in. This can be exploited by passing the following string as the token (the injection is highlighted in bold):

"random_token**\r\nset my_crafted_token 0 3600 4\r\nroot\r\n**"

When this string is passed as a token, the database will be checked for the existence of `random_token` and will add `my_crafted_token` with the value root. Later, attackers can send `my_crafted_token` and will be recognized as root.

Other instructions might be injected using this technique:

```
incr <Key> <Amount>
decr <Key> <Amount>
delete <Key>
```

where `incr` is used to increment a key, `decr` is used to decrement a key, and `delete` is used to delete a key. Attackers can also use these three functions with their `key` parameter in the same manner as the `set` and `get` functions.

Attackers can perform the same injections using the multiple-item functions: `deleteMulti`, `getMulti`, and `setMulti`, where the injection should occur in one of the `key` fields.

The CRLF injection can be used to concatenate several `get` requests. In a test we conducted, the maximum value of such concatenation was 17, including the original `get` key. The result that returns from such injection is the first key that has an associated value.

This driver vulnerability was fixed in PHP 5.5 but unfortunately exists in all prior PHP versions. According to W3Techs.com statistics on PHP versions of websites in production, more than 86 percent of PHP websites use a version older than 5.5, which means they're vulnerable to this injection if they use Memcached.

Cross-Origin Violations

Another common feature of NoSQL databases is that they can often expose an HTTP REST API that enables database query from client applications. Databases that expose a REST API include MongoDB, CouchDB, and HBase. The exposure of a REST API enables simple exposure of the database to applications—even HTML5 only–based applications—because it terminates the need for a mediate driver and lets any programming language perform HTTP queries on the database. The advantages are clear, but does this feature come with a security risk? We answer this in the affirmative: the REST API exposes the database to CSRF attacks, letting attackers bypass firewalls and other perimeter defenses.

As long as a database is deployed in a secure network behind security measures such as firewalls, to compromise the database, attackers must either find a vulnerability that will let them into the secure network or perform an injection that will let them execute arbitrary queries. When a database exposes a REST API inside the secured network, anyone with access to the secured network can perform queries on the database using HTTP only, thus allowing such queries to be initiated from the browser. If attackers can inject an HTML form into a website or trick users into the attackers' own website, they can perform any `post` action on the database by submitting the form. `Post` actions include adding documents.

In our research, we inspected Sleepy Mongoose, a full-featured HTTP interface for MongoDB. The Sleepy Mongoose API is defined by the URL as `http://{host name}/{db name}/{collection name}/{action}`. Parameters for finding a document can be included as query parameters, and new documents can be added as request data. For example, if we want to add the new document `{ username: 'attacker' }` to the collection `admins` in the database called `hr` on the `safe.internal.db` host, we would send a `post` HTTP request to `http://safe.internal.db/hr/admins/_insert` with the URL encoded data `username=attacker`.

Now let's see how a CSRF attack uses this functionality to add a new document to the `admins` collection, thus adding a new admin user to the `hr` database (which is located in the supposedly safe internal network), as Figure 5 depicts. For the attack to succeed, a few conditions must be met. First, attackers must have control over a website either of their own or from exploiting a benign, unsecured website. Attackers place an HTML form in the website and a JavaScript that will submit the form automatically, such as

```
<form action=" http://safe.internal.
db/hr/admins/_insert" method="POST"
name="csrf">
<input type="text" name="docs" value="
[{&quot;username&quot;:attacker}]" />
</form>
<script>
document.forms[0].submit();
</script>
```

Second, attackers must trick users into entering the infected site by means of phishing or inject an infection into a site that users visit regularly. Finally, users must have permissions and access to the Mongoose HTTP interface.

In this manner, attackers can perform actions—in this case, inserting new data into the database located in the internal network—without having access to the internal network. This attack is simple to execute but demands that attackers perform reconnaissance to identify the names of the host, database, and so on.

Mitigation

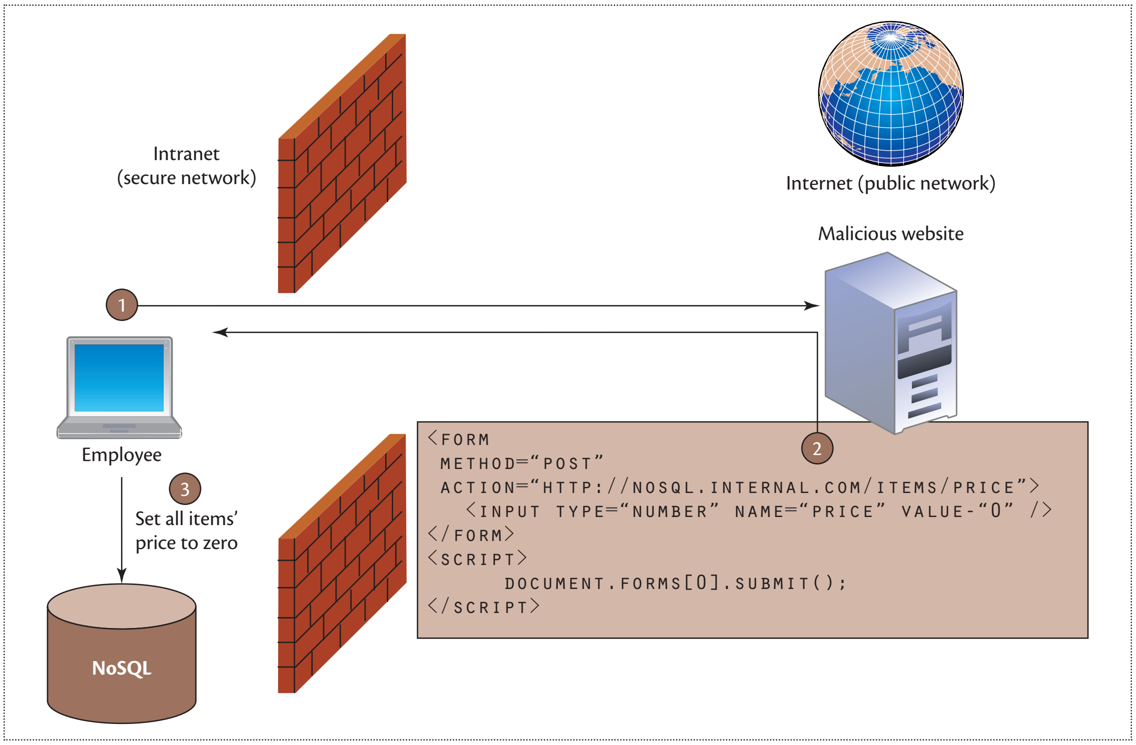
Mitigating security risks in NoSQL deployments is important in light of the attack vectors we present in this article. Unfortunately, code analysis of the application layer alone is insufficient to ensure that all threats are mitigated. Three trends make this problem even more challenging than before. First, the emerging cloud and big data systems typically execute multiple complex applications that use heterogeneous open source tools and platforms. These are commonly developed by open source communities and, in most cases, don't undergo comprehensive security testing. Another challenge is the speed of modern code development with DevOps methodologies, which aim to shorten the time between development and production. Finally, most application security testing tools can't keep up with the fast pace with which new programming languages are adopted; for instance, most security products don't support Golang, Scala, and Haskel.

Development and Testing

To fully address the threats introduced by the application layer, we need to consider the entire software development life cycle (see Figure 6).

Awareness. Obviously, building secure software that prevents injections and other potential exploits is the best and least expensive solution. It's recommended that those involved in the software life cycle receive appropriate security training for their role. For example, a developer who already understands weaknesses is less likely to introduce one into the software.

Design. An application's security aspects must be thought of and defined in the early development stages. Defining what needs to be protected in the application and how this will occur ensures this functionality is translated to tasks in the development phase and receives the right amount of attention.



```
<FORM
 METHOD="POST"
 ACTION="HTTP://NOSQL.INTERNAL.COM/ITEMS/PRICE">
   <INPUT TYPE="NUMBER" NAME="PRICE" VALUE-"0" />
</FORM>
<SCRIPT>
      DOCUMENT.FORMS[0].SUBMIT();
</SCRIPT>
```

Figure 5. Diagram of a cross-site request forgery attack on a NoSQL HTTP REST API. A user inside an internal network behind a firewall is tricked into visiting a malicious Internet page, which causes unwanted execution of queries in the NoSQL database's REST API in the internal network.

Best practices for code. Utilizing shared libraries that have undergone a security validation process, thus narrowing the window of security mistakes, is recommended. For example, using well-validated libraries for encryption reduces the risk of developers implementing encryption on their own and introducing vulnerabilities into an algorithm. Another example is the use of sanitization libraries. All injection attacks are a result of poor sanitization. Using a well-tested sanitization library greatly reduces the risk of introducing gaps in a self-developed sanitization method.

Privilege isolation. In the past, NoSQL didn't support proper authentication and role management.[9] Today, managing proper authentication and role-based access control authorization on most popular NoSQL databases is possible. These mechanisms are important for two reasons. First, they allow enforcement of the principle of least privilege, thus preventing privilege escalation attacks by legitimate users. Second, similarly to SQL injection attacks,[10] proper privilege isolation can minimize the damage in the case of data store exposure via the injections we describe in this article.

Security scanning. Running dynamic and static application security testing (DAST and SAST, respectively) on the application or source code to find injection vulnerabilities is recommended. The problem is that many tools in the market today lack rules for detecting NoSQL injections. The DAST methodology is considered more reliable than SAST,[11] especially if used with a back-end inspection technology that improves detection reliability—a methodology referred to as interactive application security testing.[9,12] It's also recommended to integrate these scans into the continuous build and deployment systems such that they run every cycle or check-in, and bugs are captured and fixed immediately—not just during the security testing phase.

This might reduce the effort of fixing security bugs for two reasons. First, the cost of fixing a bug in the development phase is much cheaper than later in the life cycle, especially because security testing tends to occur after functional testing, and fixing security bugs might introduce the need to repeat the functional testing. Second, developers might learn from their bugs in an early stage and not repeat them in similar places in later code development.

Security testing. A professional security tester should test the application. These tests should validate that all the security requirements have been met as were defined

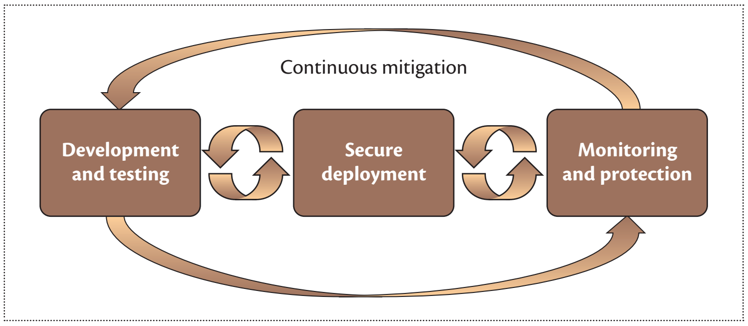


Figure 6. Life cycle of application and deployment security. To address fully the threats introduced by the application layer, we need to consider the entire software development life cycle.

in the design phase and should include penetration testing on the application and hosting infrastructure, which is recommended to resemble the production infrastructure as much as possible.

Secure Deployment

An important part of protecting the application is ensuring a secure deployment. Efforts invested in securing the application code might be wasted if the deployment is insecure. This stage is sometimes overlooked.

Network isolation. The concept of a secure internal network has been invalidated in countless attacks on enterprises such as the Adobe password breach, RSA Security, and Sony. The internal network is bound to be infiltrated at some point, and it's our duty to make it as difficult as possible for attackers to gain advantages from that point on. This is especially true for some NoSQL databases that are relatively new and lack role-based permissions, which means anyone can execute anything on them (as was the case for Memcached). For this, a strict network configuration is recommended to ensure that the database is accessible only to relevant hosts, such as the application server.

Protection of APIs. To mitigate the risks of REST API exposure and CSRF attacks, there's a need to control the requests, limiting their format. For example, CouchDB has adopted some important security measures that mitigate the risk resulting from an exposed REST API. These measures include accepting only JSON in content type. HTML forms are limited to URL-encoded content type, so attackers can't use HTML forms for CSRF. Another alternative is using Ajax requests, which are blocked by the browser thanks to the same origin policy. It's also important to ensure JSONP (JSON with padding) and cross-origin resource sharing are disabled in the server API, so no actions can be made directly from a browser. Some databases, such as MongoDB, have many third-party REST APIs; many lack the security measures we describe here.

Monitoring and Attack Detection

Humans are error prone; even after following all the secure development best practices, vulnerabilities might still be found in the software after deployment. In addition, new attack vectors might be found that were unknown at the time of development testing. Therefore, monitoring and defending the application at runtime is recommended. Although such systems might be good at finding and blocking certain attacks, they aren't aware of the application logic and the rules under which the application is supposed to work, and they won't find 100 percent of the vulnerabilities.

Web application firewalls. Web application firewalls (WAFs) are security tools that inspect HTTP data streams and detect malicious HTTP transactions. They can be implemented as appliances, network sniffers, proxies, or webserver modules and are specifically designed to provide an independent security layer for Web applications, detecting attacks such as SQL injections. Although it's known that attackers can bypass WAFs,[13] we advocate adding rules for detecting NoSQL injections to these systems as well.

Intrusion detection systems. Similar to firewalls that can detect attacks at the network level, host-based intrusion detection systems (HIDSs) guard the execution of the application and workloads on servers. HIDSs typically learn an application's normal behavior and provide alerts of activities that don't conform to the expected behavior, which can point to an attack. Such tools can detect vulnerabilities that propagate to the OS but won't be relevant for a SQL injection or CSRF attack.

Data activity monitoring. Database activity monitoring tools became a common requirement for organizations' data protection. They control access to databases, monitor activities with customizable security alerts, and create auditable reports of data access and security events. Although most solutions target relational databases, initial solutions to monitoring NoSQL data stores have already started to appear.[10] We hope that these will continue to improve and will become a common practice for NoSQL activity monitoring. These tools are the most useful monitoring and detection systems relevant for injection attacks, as we demonstrate in this article.

SIEM systems and threat intelligence. Security information and event management (SIEM) systems aggregate and correlate logs to help attack detection.

Furthermore, threat intelligence tools can assist in providing data on malicious IP addresses and domains as well as other indicators of compromise, which can help detect injections.

Runtime application self-protection. Runtime application self-protection (RASP) introduces a new application security approach in which the defense mechanism is embedded into the application at runtime, allowing it to monitor itself. The benefit of RASP over other security technologies lies in its ability to inspect the flow of the program and data being processed internally. Placing inspection points at key positions in the application allows detecting more issues with higher accuracy. On the down side, RASP takes a toll on performance, is tightly coupled with the programming language, and might break the application in production.

NoSQL databases suffer from the same security risks as their SQL counterparts. Some low-level techniques and protocols have changed, but the risks of injection, improper access control management, and unsafe network exposure remain high. We recommend using mature databases with built-in security measures. However, even using the most secure data store doesn't prevent injection attacks that leverage vulnerabilities in the Web applications accessing the data store. One way to prevent these is via careful code examination and static analysis. However, these are difficult to conduct and might have high false-positive rates. Although dynamic analysis tools were shown to be useful in detecting SQL injection attacks,[9] they should be adjusted to detect the specific NoSQL database vulnerabilities that we describe in this article. In addition, monitoring and defense systems that are relevant to NoSQL risks should be used. ■

References

1. *Imperva Web Application Attack Report*, 4th ed., Imperva, 2013; www.imperva.com/docs/HII_Web_Application_Attack_Report_Ed4.pdf.
2. *State of Software Security Report*, Varacode, 2013; www.veracode.com/blog/2013/04/changing-the-future-state-of-software-security-report-2013.
3. A. Lane, "No SQL and No Security," blog, 9 Aug. 2011; www.securosis.com/blog/nosql-and-no-security.
4. L. Okman et al. "Security Issues in NoSQL Databases," *Proc. IEEE 10th Int'l Conf. Trust, Security and Privacy in Computing and Communications* (TrustCom), 2011, pp. 541–547.
5. E. Sahafizadeh and M.A. Nematbakhsh. "A Survey on Security Issues in Big Data and NoSQL," *Int'l J. Advances in Computer Science*, vol. 4, no. 4, 2015, pp. 2322–5157.
6. M. Factor et al. "Secure Logical Isolation for Multi-tenancy in Cloud Storage," *Proc. IEEE 29th Symp. Mass Storage Systems and Technologies* (MSST), 2013, pp. 1–5.
7. "Security," *MongoDB 3.2 Manual*, 2016; http://docs.mongodb.org/manual/core/security-introduction.
8. I. Novikov, "The New Page of Injections Book: Memcached Injections," *Proc. Black Hat USA*, 2014; www.blackhat.com/docs/us-14/materials/us-14-Novikov-The-New-Page-Of-Injections-Book-Memcached-Injections-WP.pdf.
9. J. Williams, "7 Advantages of Interactive Application Security Testing (IAST) over Static (SAST) and Dynamic (DAST) Testing," blog, 30 June 2015; https://www.contrastsecurity.com/security-influencers/9-reasons-why-interactive-tools-are-better-than-static-or-dynamic-tools-regarding-application-security.
10. K. Zeidenstein, "Organizations Ramp up on NoSQL Databases, but What about Security?," blog, 1 June 2015; https://securityintelligence.com/organizations-ramp-up-on-nosql-databases-but-what-about-security.
11. V. Haldar, D. Chandra, and M. Franz, "Dynamic Taint Propagation for Java," *Proc. IEEE 21st Computer Security Applications Conf.*, 2005, pp. 303–311.
12. S.M. Kerner, "Glass Box: The Next Phase of Web Application Security Testing?," blog, 3 Feb. 2012; www.esecurityplanet.com/network-security/glass-box-the-next-phase-of-web-application-security-testing.html.
13. I. Ristic, "Protocol-Level Evasion of Web Application Firewalls," *Proc. Black Hat USA*, 2012, https://media.blackhat.com/bh-us-12/Briefings/Ristic/BH_US_12_Ristic_Protocol_Level_Slides.pdf.

Aviv Ron is a security researcher for the IBM Cyber Security Center of Excellence. His research interests include application security, specifically in cloud environments. Ron has a BSc in computer science from Ben Gurion University. Contact him at aviv1ron1@gmail.com.

Alexandra Shulman-Peleg is a cloud security domain lead in the innovation center of Citibank. During the preparation of this article, she was a senior research scientist at the IBM Cyber Security Center of Excellence. Her research interests include cloud security. Shulman-Peleg has a PhD in computer science from Tel Aviv University. She has more than 30 scientific publications in leading journals, conferences, and books. Contact her at shulman.peleg@gmail.com.

Anton Puzanov is a security researcher for the IBM Cyber Security Center of Excellence. His research interests include application security testing products. Puzanov has a BSc in communication systems engineering from Ben Gurion University. Contact him at antonp@il.ibm.com.